Gyrokinetic Particle Codes at Exascale: Challenges and Opportunities

Stéphane Ethier

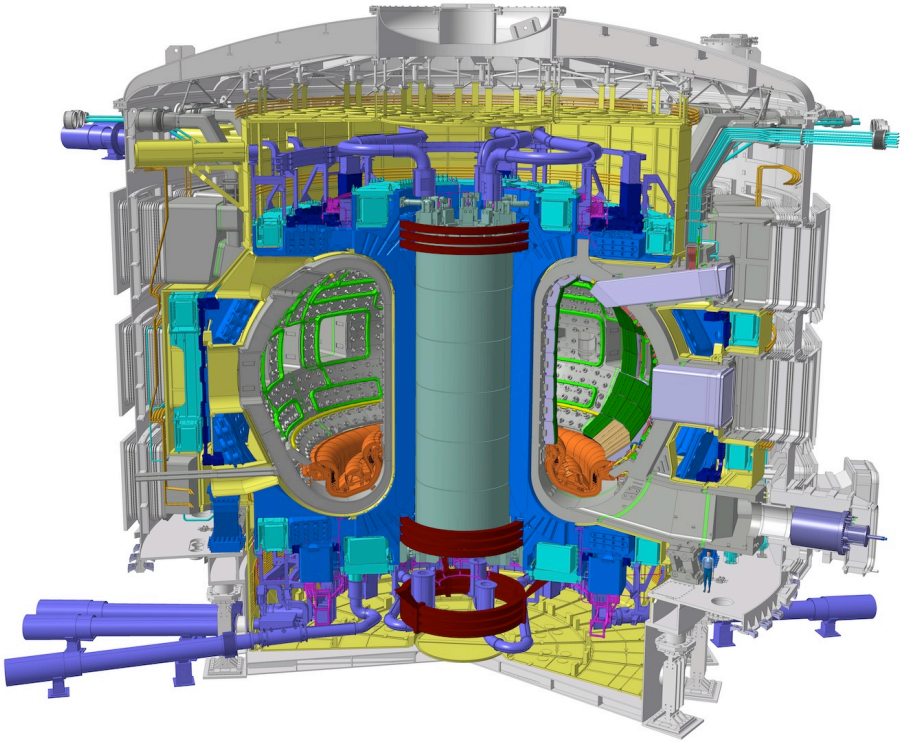
Princeton Plasma Physics Laboratory

PASC16 Conference, Lausanne, Switzerland

June 8-10, 2016

Background

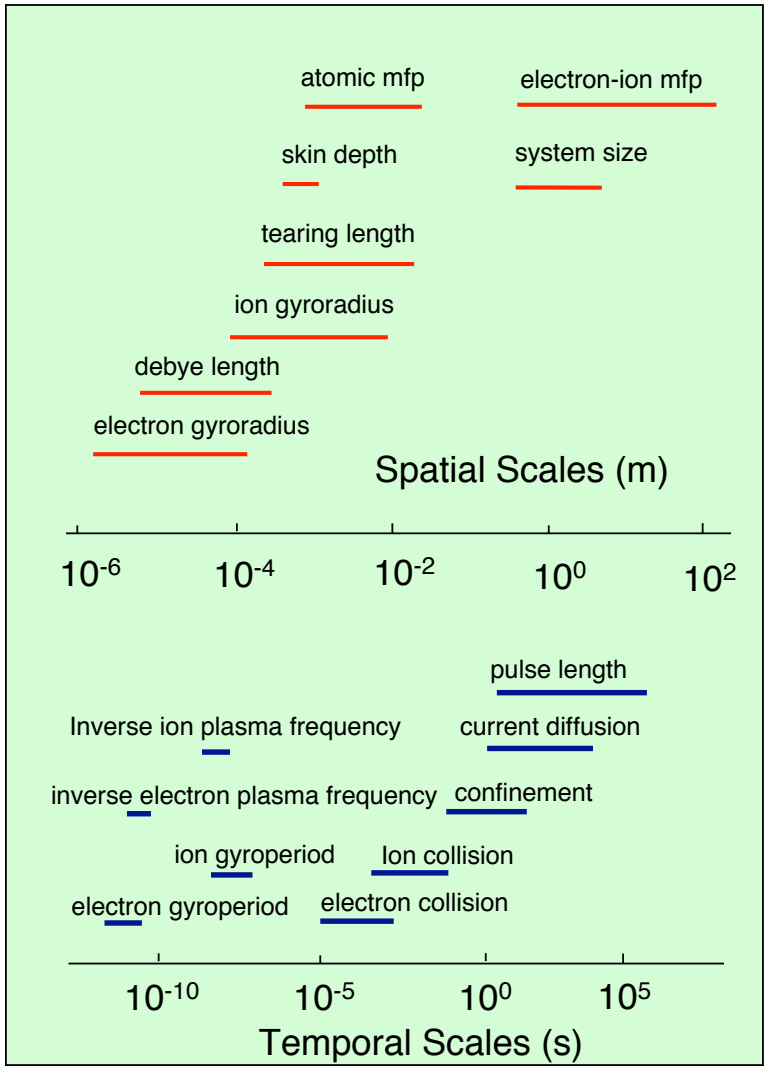
- Simulations of fusion-relevant plasmas confined in a torus-shaped magnetic bottle called “tokamak”
- “Fusion-relevant” means plasma temperatures of 100 million+ degrees
- Biggest challenge: **ITER**
  - Much larger than current tokamaks
  - “Burning plasma” experiments
  - Will require prediction of each tokamak shot
- Ultimate goal is “whole device modeling” (WDM)



PPPL

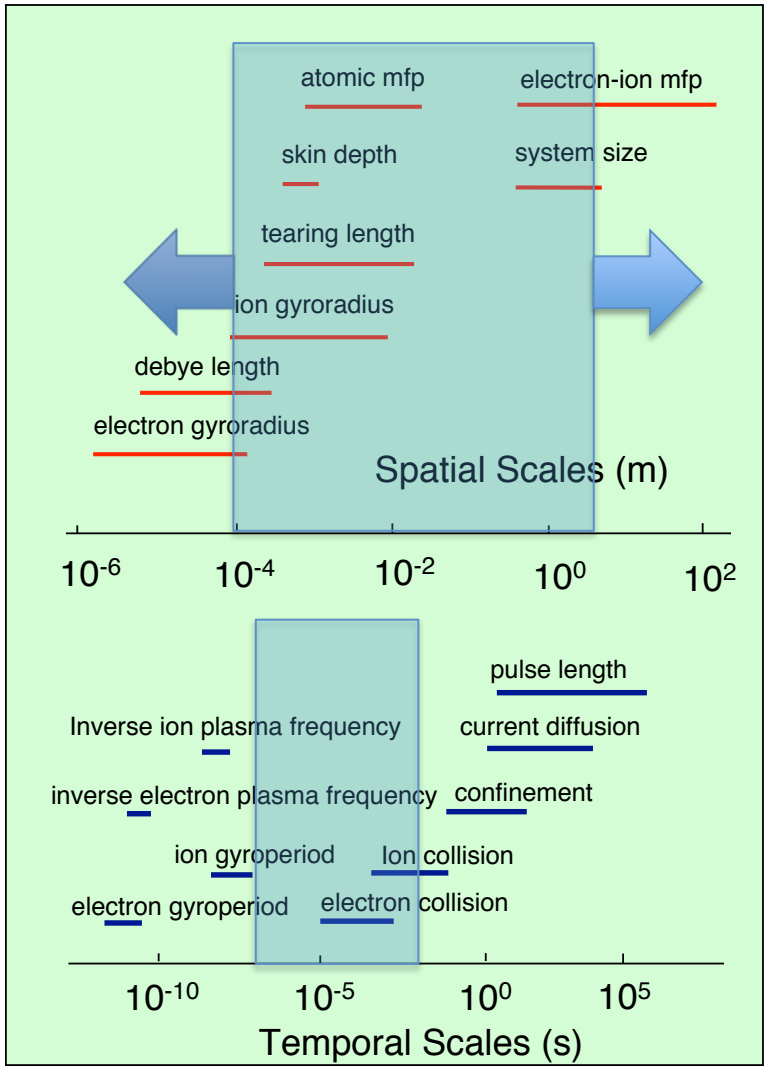
Whole device modeling is extremely challenging

- The important physics spans huge range of spatial and temporal scales
- Overlap in scales often means strong (simplified) ordering not possible
- Transport codes use fluid approach with “reduced” models to simulate full discharge (few seconds)
  - Can miss some important physics
  - Difficult to find enough parallelism



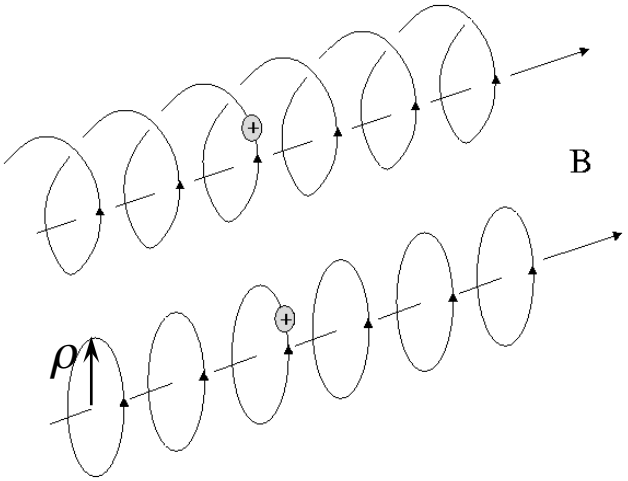
The Gyrokinetic approach

- Start in the middle and do first principles calculation
- Try to expand on “both sides”
- Kinetic approach naturally capture the important physics (e.g. turbulence)
- The particle-in-cell algorithm is currently the most promising approach to achieve exascale level
  - Lots of parallelism (200 billion particles, 100 million grid points to simulate ITER)

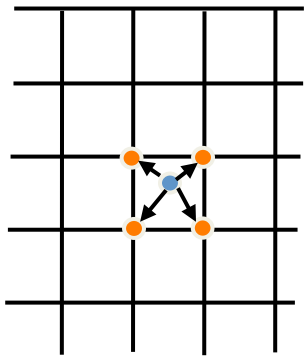


What is the "Gyrokinetic" approach anyway?

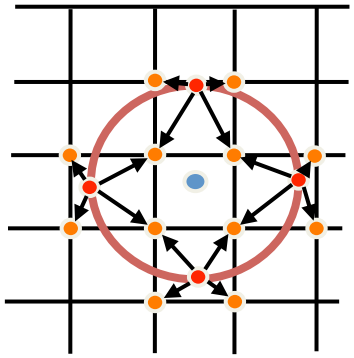
- Fast helical motion of charged particles in strong magnetic field integrated out in gyrokinetic equation
  - Helical motion replaced by moving rings of dynamically changing radius
  - No need to resolve the helical (cyclotron) motion = **larger time step**
- Complicates charge deposition step
  - Random access to memory unless particles are sorted



Charge Deposition Step (SCATTER operation)



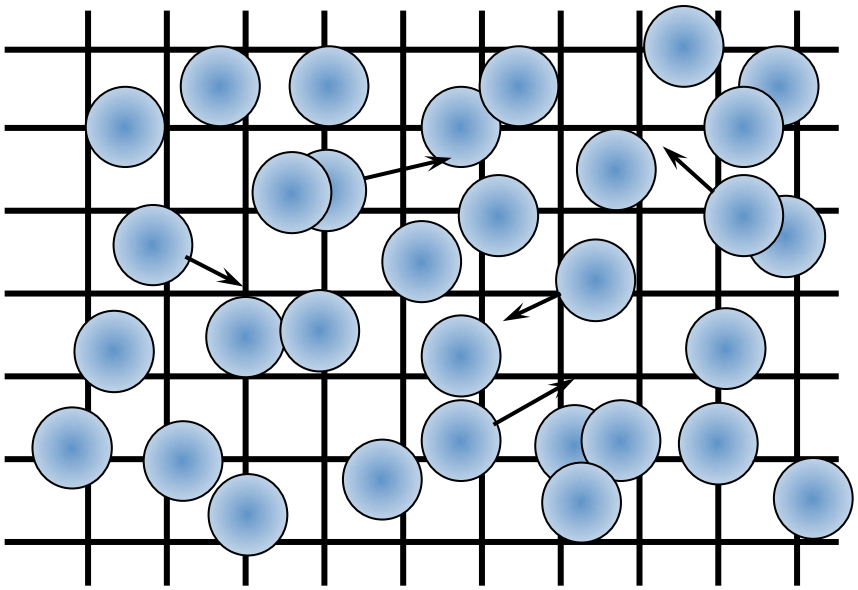
Classic PIC



4-Point Average GK
(W.W. Lee, JCP 1987)

The “low frequency” GKPIC method

- Particles sample distribution function of “gyrocenters”
- Interactions via a grid on which charges are accumulated and fields are evaluated (PDE solve) (avoids N^2 calc.)
- **100-1000X more particles than grid points**
- Grid resolution dictated by the gyroradius (Larmor radius)



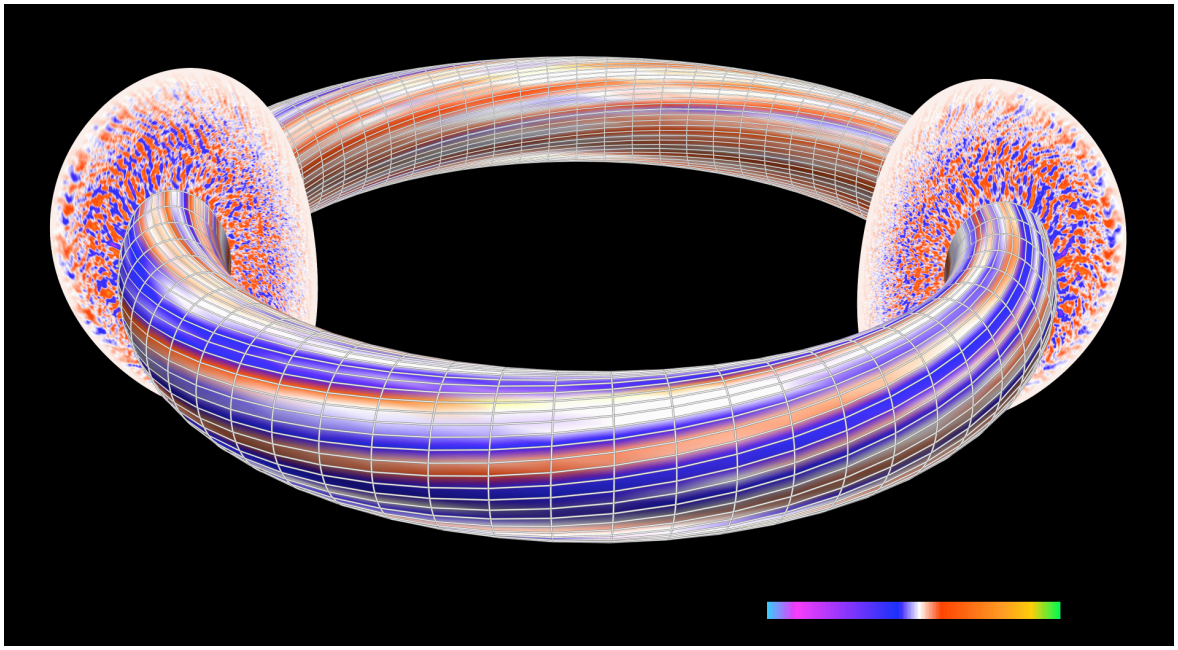
The PIC Steps

- **“SCATTER”, or deposit, charges on the grid (nearest neighbors of 4 gyro points)**
- **Solve Poisson equation**
- **“GATHER” forces on each gyrocenter particle**
- **Move particles (PUSH)**
- **Repeat…**

PPPL

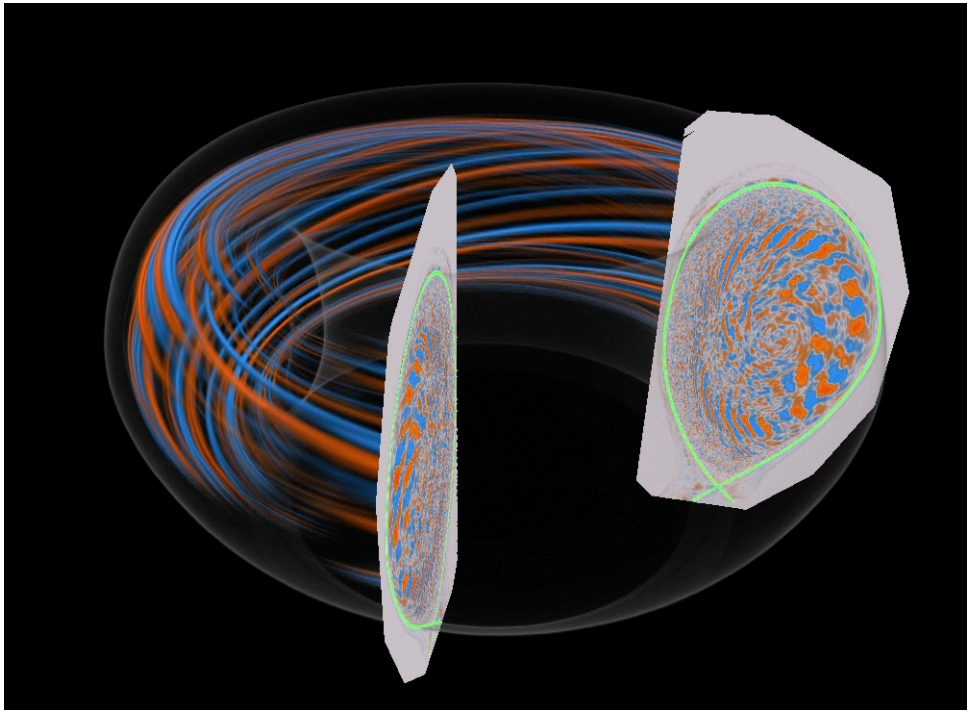
GTS global delta-f GKPIC code in general geometry for studying core turbulence in real experiments

- Efficient field-aligned grid
- Magnetic coordinates
- 4-point average method for charge deposition
- Non-spectral Poisson solver
- Energy and momentum conserving guiding center Lagrangian equations for particles time-stepping



XGC1 full-F code for plasma edge turbulence in diverted geometry, with X-point and wall

- Comprehensive first principles code
- Move particles along the characteristics ➔ODE equation (RK4)
- Solve the self-consistent field ➔ PDE equation solved using PETSc (preconditioner Hypre, KSP GMRES)
- Kinetic ions + electrons (real mass ratio)



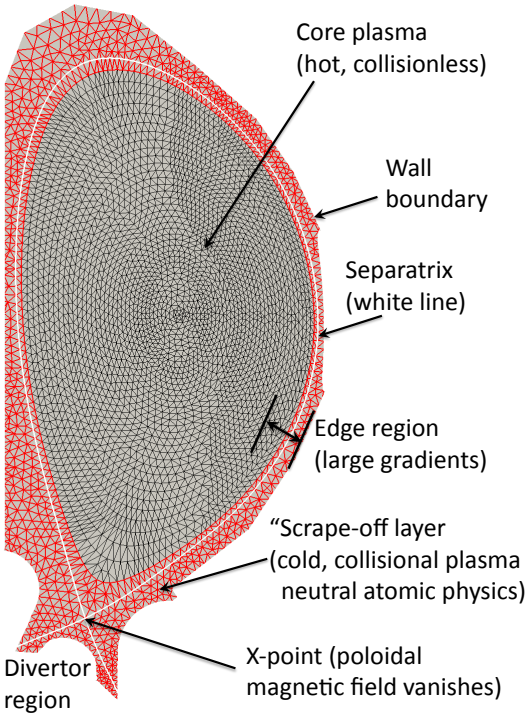
- Non spectral solver (FE iterative multi-grid solver)

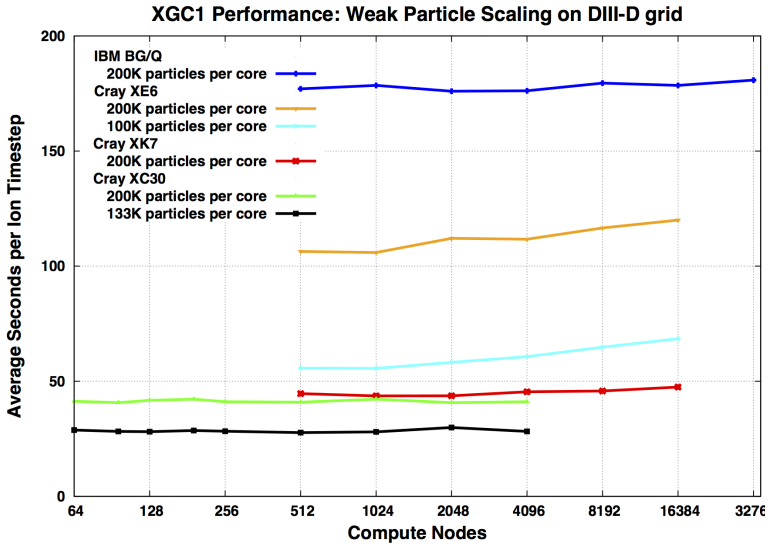
http://epsi.pppl.gov/

PPPL

XGC1's programming models

- MPI + OpenMP + CUDA Fortran for GPU
- Top level = MPI
  - Toroidal domain decomposition
  - Multi-process particle distribution within toroidal domains (require careful load balance)
  - Grid-based solver splits matrices between processes within each toroidal domain (finite element solver), implemented with PETSc library
- Fine-grained OpenMP at loop level
- CUDA Fortran for electron time-advance on GPU

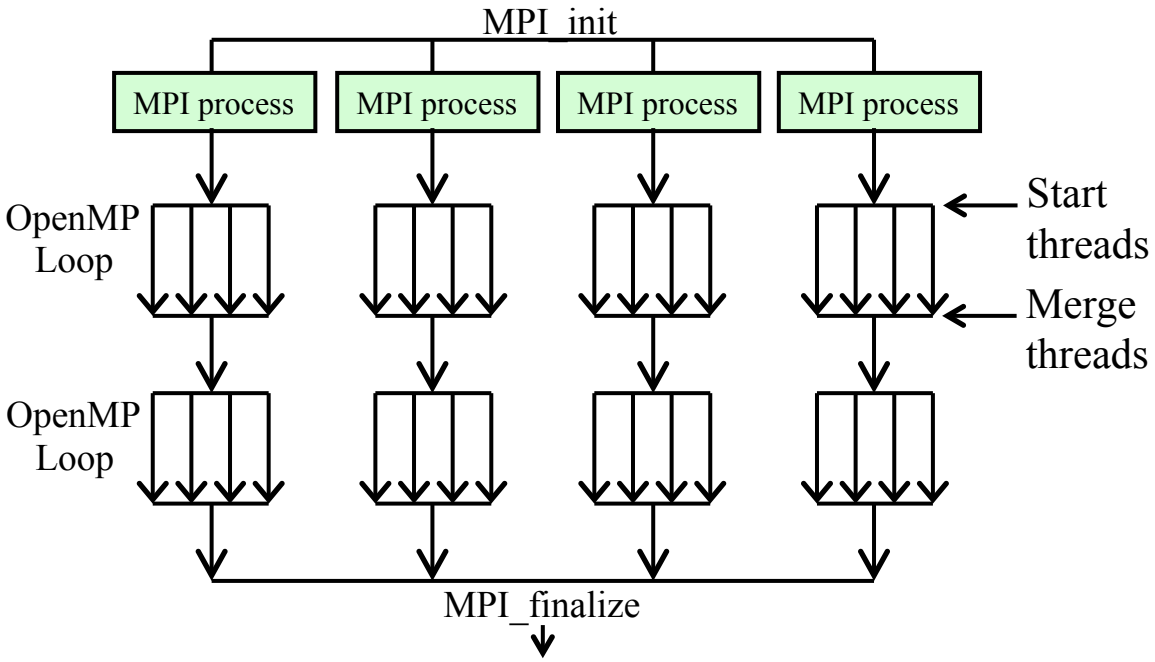




Requirements for exascale

- MASSIVE amount of parallelism
  - Billions of threads may be required to keep the hardware busy
- Vectorize your loops
  - Vectorization is back and is necessary to achieve good loop-level performance
- Having to deal with several levels of memory hierarchy
  - The way data is accessed is critical
- Avoid global synchronization
  - Having all execution threads (or MPI tasks) stop and wait for each other is a bad idea
- Limit I/O, deal with resiliency, etc.

At exascale, vectorization is important so loop-level OpenMP parallelism is too "fine-grained" →competes with vectorization



Vectorization is important even on current systems and certainly on the ones coming up (Intel Xeon Phi and GPUs)

- Bring OpenMP to a higher (coarser) level and limit "serial" regions. (At least bring to outer loop if nested loops)
- Use vectorization at the (inner) loop-level (splitting a loop in 2 levels is also a possibility

PPPL

Ways to implement coarse-level OpenMP

- MPI style

```
!$omp parallel private(thid,num_th)
num_th=omp_get_num_threads()
thid=omp_get_thread_num()
… use “thid” to divide work and do lots of it
!$omp end parallel
```

- Use OpenMP tasks!

```
!$omp parallel private(…)
!$omp master (or single)
   !$omp task
    ... Independent tasks
   !$omp end task
!$omp end master
!$omp end parallel
```

Master thread constructs a list of tasks.
Threads will be assigned tasks by scheduler
This allows for asynchronous execution
= good load balance

PPPL

How to deal with heterogeneity and many levels of memory hierarchy

- Let the computer scientists figure it out!
  - Hide this complexity from the application scientists as much as possible, while still providing a practical set of abstractions to exploit data locality and key hardware features (would be ideal!)
  - Domain Specific Languages (DSLs) and autotuning are certainly good approaches for dealing with widely used algorithms that are well-known (PDE solvers, FFT spectral solvers, etc.)
  - Projects such as Kokkos (Sandia) and Raja (LLNL) are examples
  - At the very least we need efficient thread-safe libraries, in particular, MPI library
- In reality, we will probably need to do it ourselves
  - It's all about the data structures and the way they are accessed
  - Need to worry about such things as "Array-of-Structures" (AoS) and "Structures-of-Arrays" (SoA) and even AoSoA, etc.

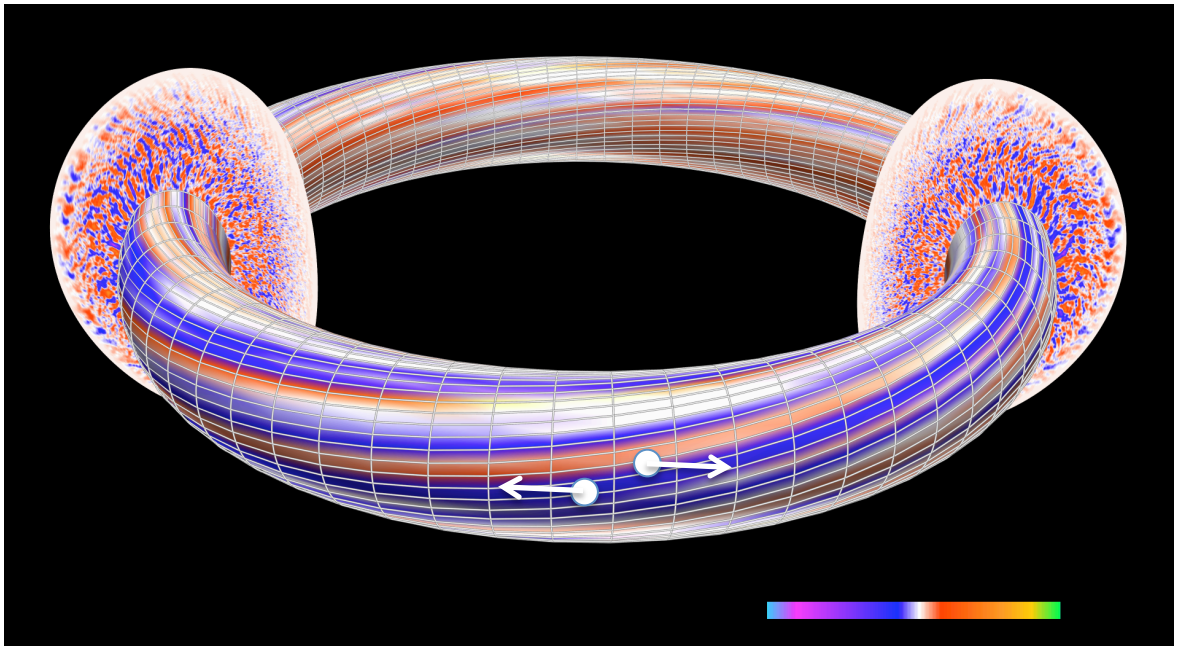
Synchronization is bad! What to do

Specific example from global GK PIC code like XGC1:

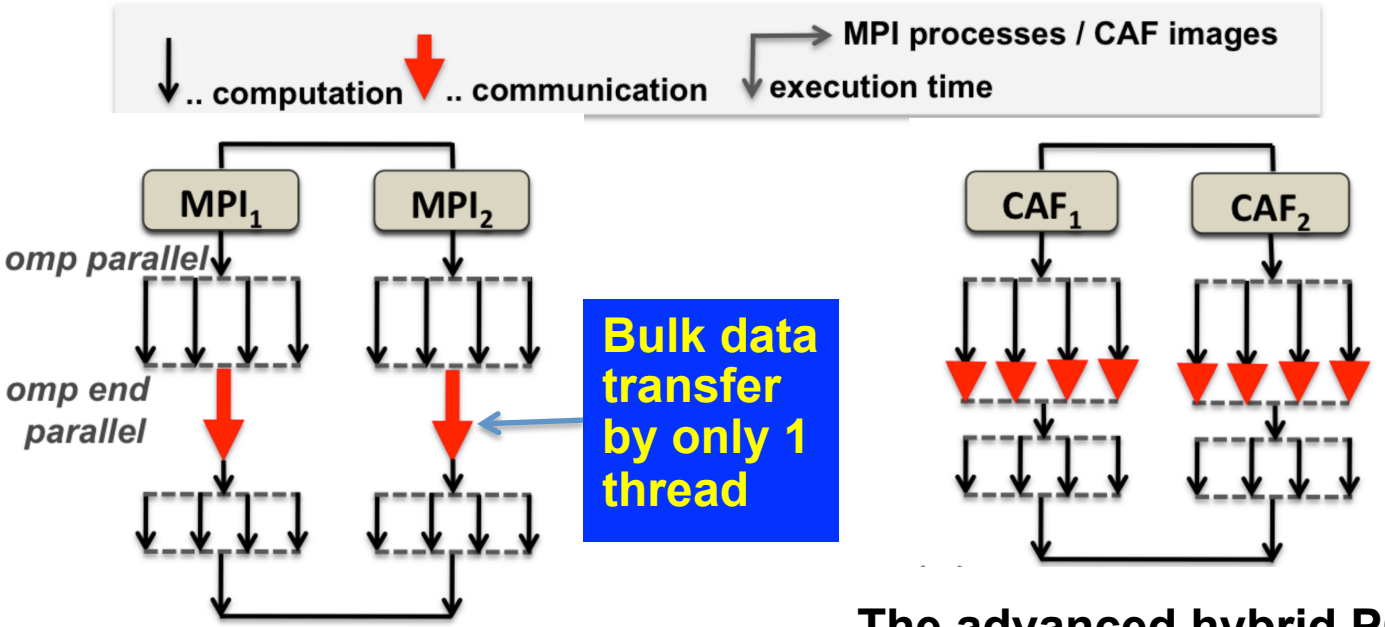
- Coherent access (or atomic updates) to large working sets
  - All threads within a process may need shared random read-modify-write access to O(100MB), the size of a poloidal plane (PIC “scatter” phase)
  - The biggest gap in existing models is the performance of fine-grained (increment 2 doubles) **atomics or transactions** (the most natural ways of expressing this). This performance deficiency motivates complex decompositions or replications. The former require inter-process communication. The latter require extra memory. Both are unattractive in the manycore limit but that’s all we have at the moment

How about inter-node communication? Domain decomposition in the toroidal direction

- The information about a particle must be moved along with that particle when exiting a toroidal domain
- This is done in a “ring-like” fashion with MPI_sendrecv (“shift” step)



Moving particles between tasks: MPI *vs.* PGAS Fortran Co-arrays (CAF)



Classical hybrid MPI/OpenMP programming model

The advanced hybrid PGAS/ OpenMP algorithm builds on the strategy of communicating threads, but allows ALL OpenMP threads per team to make communication calls to the thread-safe PGAS communication layer

PPPL

Co-array Fortran constructs (to replace MPI) included in the Fortran2008 standard

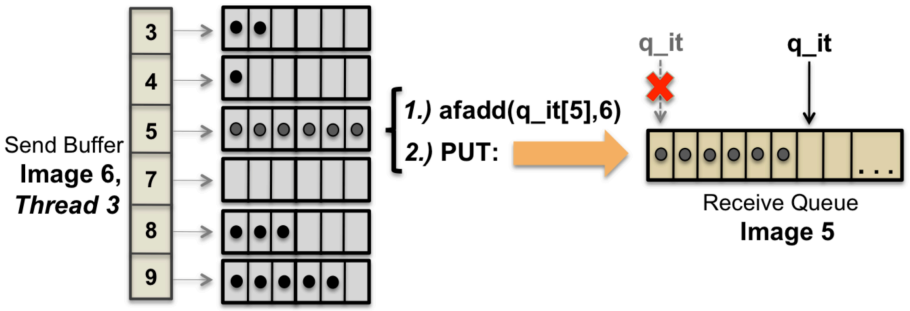
The declaration: real :: x(n)[p,q,*] means:

- An array of length n is replicated across images.
- The underlying system must build a map among these arrays.
- The logical coordinate system for images is a three dimensional grid of size(p,q,r) where r=num_images()/(pq)
- Communicating between co-array objects
  - y(:) = x(:)[p]
  - myIndex(:) = index(:)
  - yourIndex(:) = index(:)[you]
  - x(:)[q] = x(:) + x(:)[p]

CAF algorithm for overlapping communication with computation (Preissl, SC11)

```
!(1) compute shifted particles and fill the
!  receiving queues on destination images
!$omp parallel do schedule(dynamic,p_size/100)&
!$omp private(s_buf,buf_cnt) shared(recvQ,q_it)
do i=1,p_size
  dest=compute_destination(p_array(i))
  if(dest.ne.local_toroidal_domain) {
    holes(shift++)=i
    s_buf(dest,buf_cnt(dest)++)=p_array(i)
    if(buf_cnt(dest).eq.sb_size) {
      q_start=afadd(q_it[dest],sb_size)
      recvQ(q_start:q_start+sb_size-1)[dest] &
        =s_buf(dest,1:sb_size)
      buf_cnt(dest)=0  } }
enddo

!(2) shift remaining particles
empty_s_buffers(s_buf)
!$omp end parallel

!(3) sync with images from same toroidal domain
sync images([my_shift_neighbors])

!(4) fill holes with received particles
length_recvQ=q_it-1
!$omp parallel do
do m=1,min(length_recvQ,shift)
  p_array(holes(m))=recvQ(m)
enddo

!(5) append remaining particles or fill holes
if(length_recvQ-min(length_recvQ,shift).gt.0) {
  append_particles(p_array,recvQ) }
else { fill_remaining_holes(p_array,holes) }
```

Can use OMP TASK here!!!

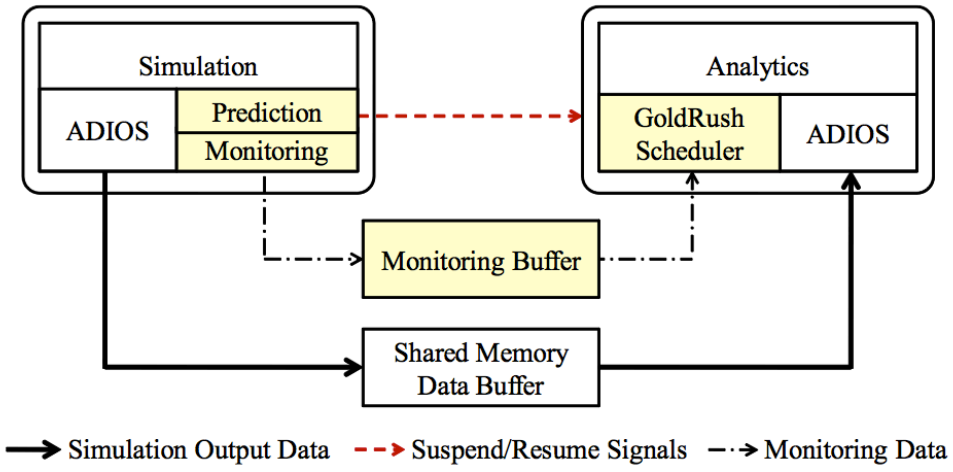


Each OMP thread goes through a subset of the particles and fills a "small" buffers with the outgoing particles until full.
At that point it uses Fortran Co-arrays to transfer the particles directly to the memory of its destination after getting the address with atomic fetch-and-add operation: AMO_AFADD()

It is hard to keep a billion threads busy at all time. Can we deal with periods of idle cores?

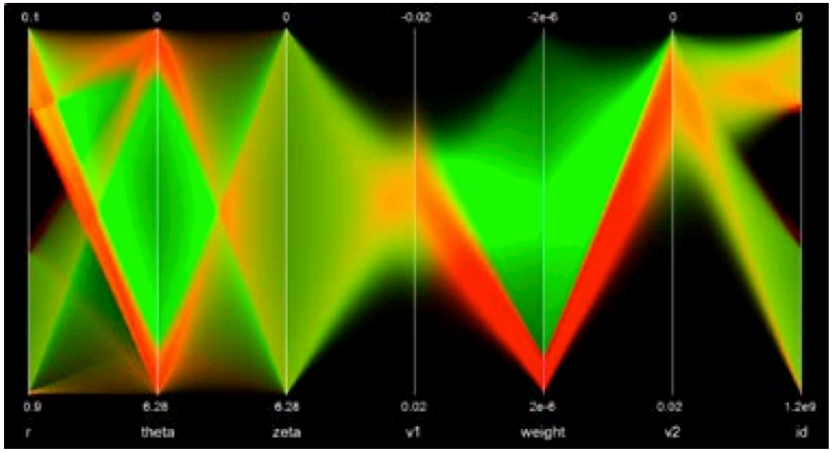
- In XGC1 runs we have **>1000 more particles than grid points**
- During computations involving only grid data (e.g. field solver) it might not efficient to engage all the cores. On heterogeneous systems, running each subroutine on both CPU and GPU is very hard. What to do?
  - Do some I/O, such as checkpointing? (needed but not at every time step)
  - Power them down to save energy? (that's boring..)
  - **Give them other tasks to do, such as diagnostics, analysis, rendering for visualization!**

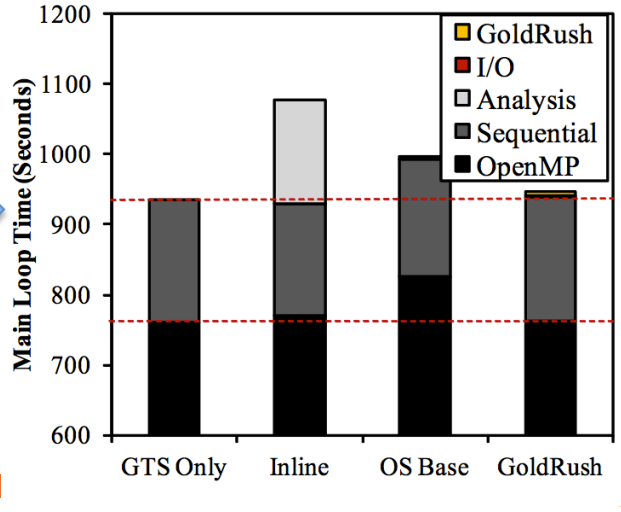
GoldRush (SC13, Zheng et al.): Monitor resources and run analyses on idle OpenMP cores



- Harvest Idle Resources for In-Situ Analytics
- Dynamically predict idle resource availability
- Reduce interference with execution throttling
- Uses **ADIOS** FlexIO method
- **Overhead of GoldRush < 0.3%**

Particle visualization (parallel coordinates)





Last comment

- There is a huge amount of parallelism in particle-in-cell codes, making them ideal for exascale
- It won't be easy though. There are many challenges starting with the indirect addressing due to the particle-grid gather-scatter operations.
- Maybe we can get rid of the grid? …

Thank you…